Finite Groundings for ASP with Functions: A Journey through Consistency

Lukas Gerlach[1] , **David Carral**[2] , **Markus Hecher**[3]
[1]Knowledge-Based Systems Group, TU Dresden, Dresden, Germany
[2]LIRMM, Inria, University of Montpellier, CNRS, Montpellier, France
[3]Massachusetts Institute of Technology, United States
lukas.gerlach@tu-dresden.de, david.carral@inria.fr, hecher@mit.edu

Abstract

Answer set programming (ASP) is a logic programming formalism used in various areas of artificial intelligence like combinatorial problem solving and knowledge representation and reasoning. It is known that enhancing ASP with function symbols makes basic reasoning problems highly undecidable. However, even in simple cases, state of the art reasoners, specifically those relying on a ground-and-solve approach, fail to produce a result. Therefore, we reconsider consistency as a basic reasoning problem for ASP. We show reductions that give an intuition for the high level of undecidability. These insights allow for a more fine-grained analysis where we characterize ASP programs as "frugal" and "non-proliferous". For such programs, we are not only able to semi-decide consistency but we also propose a grounding procedure that yields finite groundings on more ASP programs with the concept of "forbidden" facts.

1 Introduction

Answer set programming [Brewka *et al.*, 2011; Gebser *et al.*, 2012] is a logic-based formalism used in multiple fields of artificial intelligence research such as knowledge representation and reasoning but also combinatorial problem solving. State-of-the-art ASP solvers like clasp [Gebser *et al.*, 2009] or wasp [Alviano *et al.*, 2022] rely on a ground-and-solve approach. During (i) *grounding*[1] a given ASP program is instantiated with all relevant terms. Then, when (ii) *solving* the ground program, the ASP solver, which is a SAT solver extended by unfounded set propagation, excludes sets of atoms that lack foundation (i.e. unfounded sets), thereby efficiently computing answer sets. Unfortunately, with function symbols involved, already the grounding step may not terminate.

Example 1. *The program* $\{(1), (2), r(a, b)\}$ *admits exactly one answer set:* $\{r(a, b), stop(b), r(b, f(b)), stop(f(b))\}$. *Still, the grounding is infinite with terms* $b, f(b), f(f(b)), \ldots$

$$r(Y, f(Y)) \leftarrow r(X, Y), \neg stop(X). \quad (1)$$

$$stop(Y) \leftarrow r(X, Y). \quad (2)$$

[1]In the introduction, *grounding* (informally) refers to the result of a (naive) grounding procedure. We formalize this later on.

Figure 1: Wolf, Goat, Cabbage Puzzle

Such problems indeed manifest in real world applications e.g. in knowledge respresentation contexts. One prominent example is an approach for simulating sets in ASP (using function symbols) [Gaggl *et al.*, 2022]. For combinatorial problems, bounds on the size of natural numbers (which could be modelled using function symbols) are often introduced to ensure termination of the ground-and-solve approach. This is observable in many typical ASP examples.

Example 2. *We consider the famous puzzle of a farmer who needs to cross a river with a wolf, a goat, and a cabbage. They may only take one item at a time and must not leave the wolf and the goat or the goat and the cabbage alone since then the former will eat the latter (see Figure 1). One essential part of the considered modelling is a rule as the following together with enough generated atoms, e.g.* $steps(0...100)$.[2]

$$position(X, C, N+1) \leftarrow transport(X, N),$$
$$position(X, B, N), opposite(B, C), steps(N+1).$$

That is, if we guess that item X is transported in step N, then its position is updated to the opposite river bank if we are not out of steps yet. Additional rules are introduced to detect and avoid redundant positions; we elaborate on this idea later in Example 5. However, despite the redundancy check, we need to bound (guard) the term $N+1$, as otherwise the grounding is infinite. Such guards are common in ASP.

Related Work. Due to the complications with function symbols, some works avoid them altogether [Marek and Remmel, 2011]. However, some existing reasoning approaches seem promising. We observe that lazy-grounding, as used by Alpha [Weinzierl *et al.*, 2020], achieves termination on more programs than ground-and-solve approaches. For example, Alpha yields the expected finite answer set in Example 1 but still fails in Example 2 (without the $N+1$ guard). As an extension of ground-and-solve approaches, incremental solving have been proposed [Gebser *et al.*, 2019],

[2]Full example in [Gerlach *et al.*, 2024].

where one can increment the maximal number of steps used for grounding and interatively reground. Thereby, one could prevent the issue of infinite groundings when only finite answer sets exist. Various efforts have also gone into characterizing ASP programs into classes that e.g. yield finite groundings [Alviano *et al.*, 2012]. One particular idea defines the semi-decidable class of *finitely ground* programs including the decidable restriction of *finite domain* programs [Calimeri *et al.*, 2008]. This approach has been implemented in the DLV solver [Alviano *et al.*, 2010]. Still, we observe that (i) DLV [Calimeri *et al.*, 2017] does not (seem to) terminate on Examples 1 and 2, (without the $N+1$ guard). Grounding is active research, ranging from traditional instantiation [Kaminski and Schaub, 2021], over size estimations [Hippen and Lierler, 2021], lazy grounding [Weinzierl *et al.*, 2020], ASP modulo theory [Banbara *et al.*, 2017; Janhunen *et al.*, 2017; Cabalar *et al.*, 2020], and treewidth-based methods [Bichler *et al.*, 2020].

Contributions. We aim to improve existing reasoning techniques further in terms of termination. As a prerequisite, a better understanding of the hardness of reasoning is required.

- In Section 3, we consider consistency as our exemplary highly undecidable (Σ_1^1-complete) reasoning problem. We give easy to follow reductions that give an intuition into the cause of the high level of undecidability.
- Based on these studies, in Section 4, we characterize ASP programs by two essential causes for undecidability of reasoning and infinite groundings. Surprisingly, even if a program is *frugal* (only finite answer sets) and *non-proliferous* (only finitely many finite answer sets), we still obtain undecidability for program consistency.
- To still tackle consistency, in Section 5, we propose a semi-decision algorithm for frugal and non-proliferous programs that also terminates on many inconsistent programs. Based on the underlying idea of *forbidden* atoms, we moreover define a grounding procedure that terminates in more cases; like the above examples.

2 Preliminaries

We assume familiarity with propositional satisfiability (SAT) [Kleine Büning and Lettman, 1999; Biere *et al.*, 2009], where we use clauses, formulas, and assignments as usual.

Ground Answer Set Programming (ASP). We follow standard definitions of propositional ASP [Brewka *et al.*, 2011; Janhunen and Niemelä, 2016]. Let ℓ, m, and n be non-negative integers with $0 \leq \ell$, $0 \leq m \leq n$, and let $b_1, \ldots, b_\ell, a_1, \ldots a_n$ be propositional atoms. Moreover, a *literal* is an atom or its negation. A *ground rule* r is an implication of the form $b_1, \ldots, b_\ell \leftarrow a_1, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n$ where $0 \leq \ell \leq 1$; that is, a formula with at most one atom before $\leftarrow$. For such a rule, we define $H_r = \{b_1, \ldots, b_\ell\}$ and $B_r = \{a_1, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n\}$. Note that ground rules are non-disjunctive since $|H_r| \leq 1$; for *constraints* we have $|H_r| = 0$. Moreover, for a set of literals L (such as B_r), let L^+ be the set of all positive literals in L and let $L^- = \{a \mid \neg a \in L\}$. A *(normal) ground program* is a set of ground rules (and constraints).

An *interpretation* I is a set of atoms. An interpretation I *satisfies* a ground rule r if $(H_r \cup B_r^-) \cap I \neq \emptyset$ or $B_r^+ \setminus I \neq \emptyset$; it is a *model* of a ground program $\mathcal{P}$ if it satisfies all rules is $\mathcal{P}$. For a set A of atoms, a function $\varphi : A \to \mathbb{N}$ is an *ordering* over A. Consider a model I of a ground program $\mathcal{P}$, and an ordering φ over I. An atom $a \in I$ is *proven (justified)* if there is a ground rule $r \in \mathcal{P}$ with $a \in H_r$ such that (i) $B_r^+ \subseteq I$, (ii) $I \cap B_r^- = \emptyset$ and $I \cap (H_r \setminus \{a\}) = \emptyset$, as well as (iii) $\varphi(b) < \varphi(a)$ for every $b \in B_r^+$. Then, I is an *answer set* of $\mathcal{P}$ if (I) I is a model of $\mathcal{P}$, and (II) I *is proven*, i.e., every $a \in I$ is proven. Deciding whether a ground program has an answer set is called *consistency*, which is NP-complete [Marek and Truszczyński, 1991] for normal ground programs. For non-ground normal programs without functions, this is NEXPTIME-complete [Eiter *et al.*, 1994]; Σ_2^{P}-complete with bounded arities [Eiter *et al.*, 2007].

Non-Ground ASP. We define Preds, Funs, Cons, and Vars to be mutually disjoint and countably infinite sets of predicates, function symbols, constants, and variables, respectively. Every $s \in \texttt{Preds} \cup \texttt{Funs}$ is associated with some *arity* $\mathsf{ar}(s) \geq 0$. For every $i \geq 0$, both $\texttt{Preds}_i = \{P \in \texttt{Preds} \mid \mathsf{ar}(P) = i\}$ and $\texttt{Funs}_i = \{f \in \texttt{Funs} \mid \mathsf{ar}(f) = i\}$ are countably infinite. The set Terms of terms includes Cons and Vars; and contains $f(t_1, \ldots, t_i)$ for every $i \geq 1$, every $f \in \texttt{Funs}_i$, and every $t_1, \ldots, t_i \in \texttt{Terms}$. A term $t \notin \texttt{Vars} \cup \texttt{Cons}$ is *functional*. A *(ground) substitution* is partial function from variables to *ground terms*; that is, to variable-free terms. We write $[x_1/t_1, \ldots, x_n/t_n]$ to denote the substitution mapping $x_1, \ldots, x_n$ to $t_1, \ldots, t_n$, respectively. For an expression ϕ and a substitution σ, let $\phi\sigma$ be the expression resulting from ϕ by uniformly replacing every syntactic occurrence of every variable x by $\sigma(x)$ if defined.

Let $q_1, \ldots, q_\ell, p_1, \ldots, p_n \in \texttt{Preds}$ be predicates and $S_1, \ldots, S_\ell, T_1, \ldots, T_n$ be vectors over Terms. A *(non-ground) program* $\mathcal{P}$ is a set of *(non-ground) rules* of the form $H \leftarrow p_1(T_1), \ldots, p_m(T_m), \neg p_{m+1}(T_{m+1}), \ldots, \neg p_n(T_n)$ where $H = q_1(S_1), \ldots, q_\ell(S_\ell)$ with $0 \leq \ell \leq 1$, $|S_i| = \mathsf{ar}(q_i)$ for every $1 \leq i \leq \ell$, and $|T_i| = \mathsf{ar}(p_i)$ for every $1 \leq i \leq n$. Note again that we do not consider disjunctive rules. We assume that rules are *safe*; that is, every variable in a rule occurs in some T_i with $1 \leq i \leq m$. We define $\mathsf{HUniv}(\mathcal{P})$ as the set of all (ground) terms that only feature function symbols and constants in $\mathcal{P}$. For a set of ground terms T, let $\mathsf{Ground}(\mathcal{P}, T)$ be the set of ground rules containing $\rho\sigma$ for every $\rho \in \mathcal{P}$ and every substitution σ from the the variables of ρ into T. Let $\mathsf{Ground}(\mathcal{P}) = \mathsf{Ground}(\mathcal{P}, \mathsf{HUniv}(\mathcal{P}))$. Note that $\mathsf{Ground}(\mathcal{P})$ might be infinite. An *answer set* of a program $\mathcal{P}$ is an answer set of $\mathsf{Ground}(\mathcal{P})$. A ground program $\mathcal{P}_g$ is a *valid grounding* for a program $\mathcal{P}$ if $\mathcal{P}_g$ and $\mathcal{P}$ have the same answer sets.

Recap: Arithmetical/Analitical Hierarchy. We view the *arithmetical hierarchy* as classes of formal languages Σ_i^0 with $i \geq 1$ where Σ_1^0 is the class of all semi-decidable languages and Σ_{i+1}^0 results from Σ_i^0 by a Turing jump. The respective co-classes are denoted by Π_i^0. We also consider the first level of the *analytical hierarchy*, i.e. Σ_1^1 and Π_1^1, which is beyond the arithmetical hierarchy [Rogers, 1987]. These classes are merely considered via reductions to and from languages contained in or hard for the respective classes [Harel, 1986].

3 Checking Consistency of ASP Programs

In this section, we prove that the problem of checking if a program admits an answer set is highly undecidable. The upper bound follows from Lemma 1 and Proposition 1; the lower bound from Lemma 2 and Proposition 2. We include complete proofs for these lemmas in [Gerlach *et al.*, 2024].

Theorem 1. *Deciding program consistency is Σ_1^1-complete.*

Although Theorem 1 has been proven before (see Corollary 5.12 in [Marek *et al.*, 1994] and Theorem 5.9 in [Dantsin *et al.*, 2001]), we present complete proofs using (more) intuitive reductions that also lay our foundation for Section 4.

3.1 An Upper Bound for ASP Consistency

Our only goal in this subsection is to show that checking consistency is in Σ_1^1 by reduction to the following problem.

Proposition 1 ([Harel, 1986, Corollary 6.2][3])**.** *Checking if some run of a non-deterministic Turing machine on the empty word visits the start state infinitely many times is in Σ_1^1.*

For a program P and an interpretation I, let $\mathsf{Active}_I(P)$ be the set of all rules in $\mathsf{Ground}(P)$ that are not satisfied by I. If I is finite, then so is $\mathsf{Active}_I(P)$ and Active_I is computable.

Definition 1. *For a program P, let $\mathcal{M}_P$ be the non-deterministic machine that, regardless of the input, executes the following instructions:*

1. *Initialise an empty set L_0 of literals, and some counters $i := 0$ and $j := 0$.*
2. *If L_i^+ and L_i^- are not disjoint,* halt.
3. *If L_i^+ is an answer set of P,* loop *on the start state.*
4. *Initialise $L_{i+1} := L_i \cup H_r \cup \{\neg a \mid a \in B_r^-\}$ where r is some non-deterministically chosen rule in $\mathsf{Active}_{L_i^+}(P)$.*
5. *If L_i satisfies all of the rules in $\mathsf{Active}_{L_j^+}(P)$, then increment $j := j + 1$ and visit the start state once.*
6. *Increment $i := i + 1$ and go to Step 3.*

Lemma 1. *A program P is consistent iff some run of $\mathcal{M}_P$ on the empty word visits the start state infinitely many times.*

Intuitively, a run of $\mathcal{M}_P$ attempts to produce an answer set for P; if successful, it visits the start state infinitely many times. The answer set is materialised via the non-deterministic choices that instantiate $L_1, L_2, \ldots$; see Step 4.

It is important to realize that the machine $\mathcal{M}_P$ only adds proven atoms in the sequence $L_1^+, L_2^+, \ldots$ To show this, consider some $k \geq 1$ and the ordering that maps the only atom in $L_i^+ \setminus L_{i-1}^+$ to i for every $1 \leq i \leq k$. Hence, if L_k^+ is a model of P, then L_k^+ also an answer set of P and the run loops at the k-th iteration because of Step 3. Otherwise, $\mathsf{Active}_{L_k^+}(P)$ is non-empty and a rule from this set can be chosen in Step 4.

If the sequence $L_1, L_2, \ldots$ is infinite and $\mathcal{M}_P$ visits the start state infinitely many times during the considered run, then $\bigcup_{i\geq 1} L_i^+$ is a model of P. This is because, for every $j \geq 1$, there is some $i \geq j$ such that L_i^+ satisfies all of the rules in $\mathsf{Active}_{L_j^+}(P)$. Therefore, since every atom in $\bigcup_{i\geq 1} L_i^+$ is proven, this interpretation is an answer set of P.

[3] The original result shows Π_1^1-completeness for the complement.

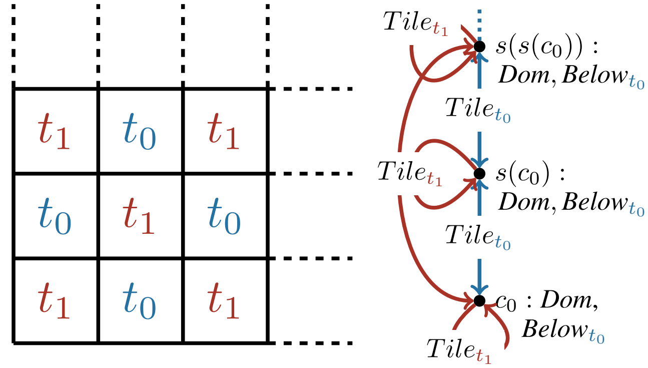

Figure 2: A Solution of $\mathfrak{X}$ and the Corresponding Answer Set of $P_{\mathfrak{X}}$

3.2 A Lower Bound for ASP Consistency

Our only goal in this subsection is to show that checking consistency is Σ_1^1-hard by reduction from the following problem.

Definition 2. *A* tiling system *is a tuple $\langle T, HI, VI, t_0\rangle$ where T is a finite set of tiles, HI and VI are subsets of $T \times T$, and t_0 is a tile in T. Such a tiling system admits a* recurring solution *if there is a function $f : \mathbb{N} \times \mathbb{N} \to T$ such that:*

1. *For every $i, j \geq 0$, we have that $\langle f(i, j), f(i + 1, j)\rangle \notin HI$ and $\langle f(i, j), f(i, j + 1)\rangle \notin VI$.*
2. *There is an infinite subset S of $\mathbb{N}$ such that $f(0, j) = t_0$ for every $j \in S$.*

Proposition 2 ([Harel, 1986, Theorem 6.4][4])**.** *Checking if a tiling system admits a recurring solution is Σ_1^1-hard.*

Condition 2 in Defintion 2 implies that, given any position in the first column, we will eventually find the special tile if we move upwards on the grid after a finite amount of steps.

Definition 3. *For a* tiling system $\mathfrak{T} = \langle T, HI, VI, t_0\rangle$*, let $P_{\mathfrak{T}}$ be the program that contains the ground atom $Dom(c_0)$ and all of the following rules:*

$$Dom(s(X)) \leftarrow Dom(X) \tag{3}$$

$$Tile_t(X, Y) \leftarrow Dom(X), Dom(Y), \{\neg Tile_{t'}(X, Y) \mid t' \in T \setminus \{t\}\} \quad \forall t \in T \tag{4}$$

$$\leftarrow Tile_t(X, Y), Tile_{t'}(s(X), Y) \quad \forall \langle t, t'\rangle \in HI \tag{5}$$

$$\leftarrow Tile_t(X, Y), Tile_{t'}(X, s(Y)) \quad \forall \langle t, t'\rangle \in VI \tag{6}$$

$$Below_{t_0}(Y) \leftarrow Tile_{t_0}(c_0, s(Y)) \tag{7}$$

$$Below_{t_0}(Y) \leftarrow Below_{t_0}(s(Y)) \tag{8}$$

$$\leftarrow Dom(Y), \neg Below_{t_0}(Y) \tag{9}$$

Lemma 2. *A tiling system $\mathfrak{T}$ admits a recurring solution iff the program $P_{\mathfrak{T}}$ is consistent*

Lemma 2 holds since each answer set of $P_{\mathfrak{T}}$ faithfully encodes a recurring solution of a tiling system $\mathfrak{T}$. We clarify this brief intuition with an example.

Example 3. *The tiling system $\mathfrak{X} = \langle\{t_0, t_1\}, HI, VI, t_0\rangle$ where $HI = VI = \{\langle t_0, t_0\rangle, \langle t_1, t_1\rangle\}$ admits two recurring solutions. The program $P_{\mathfrak{X}}$ admits two answer sets; each of them encodes a solution of $\mathfrak{X}$. One of these solutions and the*

[4] The original result shows Σ_1^1-completeness.

corresponding answer set are depicted in Figure 2. Note how the tile t_0 covers the position $\langle 0, 1\rangle$ of the positive quadrant; this is encoded by the atom Tile$_{t_0}(c_0, s(c_0))$ in the answer set.

Intuitively, Rule 3 in Definition 3 ensures that the domain of every answer set is countably infinite to provide enough space for a possible recurring solution. Rule 4 ensures that every position in the positive quadrant is covered by exactly one tile. Constraints 5 and 6 are violated if the horizontal and vertical incompatibilities are not satisfied, respectively. Formulas 7, 8, and 9 ensure that every position in the left column is below a position covered with the special tile that appears infinitely often in a valid recurring solution.

4 Frugal and Non-Proliferous Programs

In this section, we aim to develop a better understanding for why consistency has such a high level of undecidability. One particularly hard (undecidable) case to check is the existence of an infinite answer set, so it is straightforward to restrict to ASP programs that only admit finite answer sets (they might not admit any or infinitely many of these). Especially in cases, where we are not only interested in consistency but in enumerating all answer sets, it is also of interest that there is only finitely many of them.

Definition 4. *A program is* frugal *if it only admits finite answer sets; it is* non-proliferous *if it only admits finitely many finite answer sets (but arbitrarily many infinite ones).*

Not every frugal program is also non-proliferous.

Example 4. *The following ASP program admits infinitely many finite answer sets but no infinite one.*

$$next(Y, f(Y)) \leftarrow next(X, Y), \neg last(Y).$$
$$last(Y) \leftarrow next(X, Y), \neg next(Y, f(Y)).$$
$$done \leftarrow last(Y). \quad \leftarrow \neg done. \quad next(c, d).$$

Clearly, $\{next(c, d), last(d), done\}$ is an answer set. Also, any finite chain of $next$ relations terminated by $last$ is an answer set. However, an infinite $next$-chain is not an answer set as it cannot contain any $last$ atom, hence does not feature $done$, and therefore violates the constraint.

4.1 Undecidability of These Notions

Within the scope of this subsection, let P be an arbitrary program. Both of the above problems, i.e. P being frugal or non-proliferous, are undecidable and not even semi-decidable. We start with the second problem since it is comparably "easy".

Theorem 2. *Deciding if P is non-proliferous is Σ^0_2-complete.*

The previous result follows directly from Lemmas 3 and 4.

Lemma 3. *Deciding if P is non-proliferous is in Σ^0_2.*

Proof. We show first (†) that one can semi-decide for a given n if a program has at least n finite answer sets. This is possible by enumerating and checking all answer set candidates. Once the n-th answer set has been found, the procedure halts and accepts (otherwise it may run forever).

The decision problem from the lemma can now be semi-decided with an oracle for (†) as follows. Enumerate all naturals n and check for each, if P admits at least n finite answer sets (with the oracle). If yes, continue with $n + 1$; otherwise accept (since only finitely many finite answer sets exist). □

Lemma 4. *Deciding if P is non-proliferous is Σ^0_2-hard.*

Proof. Consider the universal halting problem, which is Π^0_2-hard, i.e. the check if a Turing machine halts on all inputs. We construct M' for a given TM M that on input n runs M on all inputs of length at most n. We have that M' terminates on infinitely many inputs if and only if M halts on all inputs. Hence, deciding if a TM halts on infinitely many inputs is Π^0_2-hard. Therefore, the complement, i.e. deciding if a TM halts on only finitely many inputs, is Σ^0_2-hard. We generate all (finite) inputs with an ASP program and ensure that the program has a finite answer set for a generated input iff the TM halts on that input.[5] Therefore, deciding if an ASP program only admits finitely many finite answer sets is Σ^0_2-hard. □

The Turing machine simulation utilizes a frugal program. Therefore, checking if a program is non-proliferous remains Σ^0_2-hard for frugal programs. Deciding if an ASP program is frugal on the other hand is way beyond Σ^0_2 and not even in the arithmetical hierarchy (just as checking consistency).

Theorem 3. *Deciding if P is frugal is Π^1_1-complete.*

Proof Sketch. For membership, we adjust the machine from Definition 1 to halt instead of loop when it encounters a finite answer set. Hardness follows from the same reduction as Lemma 2. To see that this holds, note that $P_{\mathfrak{T}}$ either has an infinte answer set (i.e. is not frugal) if $\mathfrak{T}$ has a solution or has no answer set at all (i.e. is frugal) if $\mathfrak{T}$ has no solution. □

4.2 Consistency Becomes (Only) Semi-Decidable

If an ASP program is frugal, consistency is semi-decidable.

Theorem 4. *Consistency for frugal programs is in Σ^0_1.*

Proof. Enumerate all answer set candidates and check if they are answer sets. If there is an answer set, then there must be a finite one so the procedure terminates in this case. □

Somewhat surprisingly, even for frugal and non-proliferous programs consistency remains undecidable. The issue is that the maximum answer set size is still unknown.

Theorem 5. *Consistency for frugal and non-proliferous programs is Σ^0_1-hard.*

Proof. We reduce from the halting problem with part of the program used for the TM simulation in Lemma 4. We omit the part that generates all possible inputs. Instead, we encode the input word with ground atoms directly. The program admits a (single) finite answer set if the machine halts on its input. Otherwise, it does not admit any answer set. In any case, the program is both frugal and non-proliferous. □

The programs in the introduction have a finite bound on the size of their answer sets; they are frugal and non-proliferous.

[5]We show the machine simulation in [Gerlach *et al.*, 2024].

5 Improved Reasoning Procedures

In this section, we describe an approach for the computation of answer sets that builds upon a basic procedure for consistency checking. For frugal programs, this is a semi-decision procedure. However, unsatisfiable programs are often not detected as such. Therefore, we improve the procedure by ignoring *forbidden* atoms that may never occur in any answer set. While it is undecidable to check if an atom is forbidden, we give a proof-of-concept algorithm for a sufficient condition. For some simplified but unsatisfiable versions of Examples 1 and 2, we argue that the sufficient condition is powerful enough to detect the essential forbidden atoms. This makes the enhanced semi-decision procedure detect them as unsatisfiable. Later on, we also propose a procedure based on the forbidden atoms idea to produce valid groundings that are finite more often compared to tranditional approaches.

5.1 Limits of Semi-Decision

Even when $\mathsf{Ground}(\mathcal{P})$ is infinite, it is arguably not hard to come up with a semi-decision procedure for consistency when only considering frugal ASP programs $\mathcal{P}$. This is the same as asking if an arbitrary program has a finite answer set. We have shown semi-decidability in Theorem 4 and we can also achieve this by modifying $\mathcal{M}_P$ from Definition 1 such that it accepts when it encounters a finite answer set instead of entering an infinite loop. While this machine resembles a lazy-grounding idea, we may also describe a semi-decision procedure that incrementally enlarges a ground program.

Definition 5. *Consider the procedure* $\mathsf{IsConsistent}(\cdot)$ *that takes a program P as input where $P_g = \mathsf{Ground}(P)$:*

1. *Initialize $i := 1$ and $A_0 := \emptyset$.*
2. *Set $A_i := A_{i-1} \cup \bigcup\{H_r \mid r \in P_g; B_r^+ \subseteq A_{i-1}\}$.*
3. Reject *if $A_i = A_{i-1}$.*
4. *Set $P_i := \{r \in P_g \mid B_r^+ \subseteq A_i\}$.*
5. Accept *if P_i has an answer set I with $\mathsf{Active}_I(P) = \emptyset$.*
6. *Set $i := i + 1$ and go to Step 2.*

Proposition 3. *Given some program P, the procedure $\mathsf{IsConsistent}(P)$ accepts (and halts) if and only if P has a finite answer set.*

Proof. If P has a finite answer set I, then pick the smallest i such that $A_i \supseteq I$. Since every atom in I is proven, the procedure does not reject up until reaching i. Moreover, since I is an answer set of P, it is an answer set of P_i; and $\mathsf{Active}_I(P) = \emptyset$. Therefore, the procedure accepts in step i.

If the procedure accepts, it does so for some i and there is a (finite) answer set I of P_i. Since $\mathsf{Active}_I(P)$ is empty, all rules in $\mathsf{Ground}(P)$ are satisfied by I. Since I is an answer set of P_i, all atoms in I are proven in P. Hence, I is a finite answer set of P. ☐

Since $\mathsf{IsConsistent}$ semi-decides consistency for frugal programs, it will necessarily yield a finite answer set for all examples from the introduction. Still, inconsistent programs $\mathcal{P}$ are rarely caught by $\mathsf{IsConsistent}$, unless $\mathsf{Ground}(\mathcal{P})$ is finite. To illustrate this, we condense the encoding of Example 2 to a simple case of detecting redundancies.

Example 5. *Consider the following program P.*

$$\mathit{fct}(a,0). \qquad \mathit{eq}(X,X) \leftarrow \mathit{fct}(X,N). \qquad \leftarrow \mathit{redundant}.$$
$$\mathit{lt}(N,s(N)) \leftarrow \mathit{fct}(X,s(N)).$$
$$\mathit{lt}(N,N') \leftarrow \mathit{lt}(N,M), \mathit{lt}(M,N').$$
$$\mathit{fct}(b,s(N)) \leftarrow \mathit{fct}(a,N). \qquad \mathit{fct}(a,s(N)) \leftarrow \mathit{fct}(b,N).$$
$$\mathit{diff}(N,M) \leftarrow \mathit{fct}(X,N), \mathit{fct}(Y,M), \neg \mathit{eq}(X,Y), \mathit{lt}(N,M).$$
$$\mathit{redundant} \leftarrow \mathit{fct}(X,N), \mathit{fct}(Y,M), \mathit{lt}(N,M), \neg \mathit{diff}(N,M).$$

Intuitively, a timeline is constructed that always flips fact (fct) a to b and vice versa in each step. We forbid redundancies, e.g. we do not want fct a in time steps say 0 and 2. This is impossible and we will always be forced to derive redundant at some point because some of the $\mathit{diff}(N,M)$ atoms cannot be proven. However, the procedure $\mathsf{IsConsistent}$ simply does not terminate here.

Note that we ran the example with Alpha, but encountered a stack overflow. This may indicate that Alpha introduces too many ground atoms. Also, (i)DLV as well as gringo/clingo do not (seem to) terminate. The $\mathsf{IsConsistent}$ check also runs into similar problems for a slight variation of Example 1.

Example 6. *Consider the following extension of Example 1.*

$$r(a,b). \qquad stop(Y) \leftarrow r(X,Y). \qquad \leftarrow r(b,f(b)).$$
$$r(Y,f(Y)) \leftarrow r(X,Y), \neg stop(X).$$

The program does not have any answer set. Furthermore, the $\mathsf{IsConsistent}$ check does not terminate.

To our surprise, Alpha captures this as unsatisfiable. Still, (i)DLV and gringo/clingo do not (seem to) terminate.

5.2 Ignoring Forbidden Atoms

We aim to extend $\mathsf{IsConsistent}$ further to capture the previous examples as unsatisfiable. We adjust the assignment of A_i in Item 2 as follows; keeping Proposition 3 intact.

2. Set $A_i := A_{i-1} \cup \{a \mid a$ not *forbidden* in P and $\{a\} = H_r$ for some $r \in \mathsf{Ground}(P)$ with $B_r^+ \subseteq A_{i-1}\}$.

To keep the A_i (and thus P_i) small, we only consider atoms that are not forbidden in P. Formally, an atom a is *forbidden* (in a program P) if a does not occur in any answer set of P. Intuitively, a forbidden atom will necessarily lead to a contradiction or it will be impossible to show that it is proven. Unfortunately, it is undecidable to check if an atom is forbidden in the formal sense, essentially because entailment of ground atoms over ASP programs is undecidable.

Proposition 4. *It is undecidable if an atom is forbidden.*

Proof. We reuse the Turing machine simulation from Theorem 5 to show a reduction from the complement of the halting problem. If the machine halts, the simulation has a single answer set with the atom Halt. Otherwise, the simulation does not admit an answer set. Hence, the machine does not halt iff Halt is forbidden. ☐

We introduce some auxiliary definitions with the aim of giving a (rather tight) sufficient condition for finding forbidden atoms. Given a rule r in P and any two interpretations L^+ and L^-, we define the following.

- $r^+(L^+, L^-)$ as the minimal interpretation that contains the single atom in $H_r\sigma$ (unless $|H_r| = 0$) for every substitution σ with $B_r^+\sigma \subseteq L^+$ and $B_r^-\sigma \subseteq L^-$; and, if $|B_r^-| = 1$, also contains $B_r^-\sigma$ for every substitution σ with $B_r^+\sigma \subseteq L^+$ and $H_r\sigma \subseteq L^-$.
- $r^-(L^+, L^-)$ is $\emptyset$ if $|B_r^+| \neq 1$; otherwise it is defined as the minimal interpretation that contains $B_r^+\sigma$ for every substitution σ with $B_r^-\sigma \subseteq L^-$ and $H_r\sigma \subseteq L^-$

We define a way of applying rules "in reverse" here and restrict these cases to $|B_r^-| = 1$ and $|B_r^+| = 1$, respectively. In principle one could relax this by considering every possible choice for the atoms from B_r^- or B_r^+. It remains for practical evaluations to determine if this is a good trade-off between simplicity and generality. For a program P and two interpretations L^+ and L^-, we define the following. For any interpretation L, let the *term-atoms* $\mathit{TA}^P(L)$ be the set of all (ground) atoms with predicates and ground terms from P, terms in L, and arbitrary constants. Also, for each sign $s \in \{+, -\}$:

- $P_0^s(L^+, L^-)$ is L^s, and
- for every $i \geq 0$, $P_{i+1}^s(L^+, L^-)$ is the minimal interpretation that contains $P_i^s(L^+, L^-)$ and, for every $r \in P$, $r^s(P_i^+(L^+, L^-), P_i^-(L^+, L^-)) \cap \mathit{TA}^P(L^+ \cup L^-)$;
- $P_\infty^s(L^+, L^-)$ is $\bigcup_{i \geq 0} P_i^s(L^+, L^-)$

We intersect with $\mathit{TA}^P(L^+ \cup L^-)$ only to keep $P_j^s(\dots)$ finite for all j (and hence for ∞); almost arbitrary extensions are possible here. Intuitively, for sets of atoms that must be true L^+ and must be false L^-, P_∞^+ and P_∞^- close these sets under certain (not all, as there might be infinitely many) inferences to obtain larger sets of atoms that must be true or false, respectively. By definition, we can show via induction that P_∞^s only makes sound inferences of atoms.

Lemma 5. *For a program P any two interpretations L^+ and L^- and any answer set I of P; if $L^+ \subseteq I$ and $L^- \cap I = \emptyset$, then $P_\infty^+(L^+, L^-) \subseteq I$ and $P_\infty^-(L^+, L^-) \cap I = \emptyset$.*

To check if an atom a is forbidden in a program P, we backtrack atoms that must be true (L^+) and must be false (L^-) to be able to prove a. To this aim, we describe a procedure with the help of some auxiliary definitions. We say that an atom *has support* in P for interpretations L^+ and L^- if there is a rule $r \in P$ and a substitution σ with $H_r\sigma = \{a\}$, $B_r^+\sigma \subseteq L^+$ and $B_r^-\sigma \subseteq L^-$. Intuitively, having support is almost like being proven but the set of non-derived atoms is given explicitly using L^-. For a rule $r \in P$ and a substitution σ, an *r-extension* of σ is a substitution σ' that agrees with σ on variables from H_r and additionally, for each variable X in r that occurs in B_r^+ but not in H_r, if X occurs in a position that can only feature constants, $\sigma'(X)$ is one of these constants; otherwise, i.e. if functional terms may occur in the position of X, $\sigma'(X)$ is a fresh constant.[6] To check if a is forbidden, we call $\textsf{IsForbidden}(P, \{a\}, \emptyset)$ from Algorithm 1. The general idea of the procedure is as follows. We first check if L^+ and L^- contradict each other. When we

[6]We can perform static analysis on the positive part of P to obtain the possible constants for each position in P and also to determine if a function symbol might occur in a position.

Algorithm 1 IsForbidden

Require: program P, interpretations L^+ and L^-
Ensure: some atom in L^+ is forbidden or $L^+ \cap L^- \neq \emptyset$

```
1:  L⁺ ← P⁺_∞(L⁺, L⁻) and L⁻ ← P⁻_∞(L⁺, L⁻)
2:  if L⁺ ∩ L⁻ ≠ ∅ then
3:      return true
4:  end if
5:  someFrbdn ← false
6:  for all a ∈ L⁺ without support do
7:      aFrbdn ← true
8:      for all r ∈ P and substitution σ and g mapping all
        fresh constant to arbitrary terms except such constants
        with H_r σ = {g(a)} do
9:          if g(a) features terms not in L⁺ ∪ L⁻ then
10:             aFrbdn ← false
11:             break
12:         end if
13:         K⁺ ← g(L⁺) and K⁻ ← g(L⁻)
14:         for all r-extensions σ' of σ do
15:             J⁺ ← K⁺ ∪ (B⁺_r σ' ∩ TA^P(K⁺ ∪ K⁻)) and
                J⁻ ← K⁻ ∪ (B⁻_r σ' ∩ TA^P(K⁺ ∪ K⁻))
16:             aFrbdn ← aFrbdn ∧ IsForbidden(P, J⁺, J⁻)
17:         end for
18:     end for
19:     someFrbdn ← someFrbdn ∨ aFrbdn
20: end for
21: return someFrbdn
```

reach this base case, we know that our initial atom a must be forbidden. Otherwise, we check if there is an unproven atom a left in line 6. If so, we check all ways in which a could be proven in line 8. If a can potentially be proven with some unknown function symbols, we assume a not to be forbidden in line 9. Otherwise, we perform recursive calls to IsForbidden within the loop in line 14 such that a is marked as not forbidden if at least one recursive call does not involve forbidden atoms or a contradiction. This means, a might be provable. By a recursive analysis of the algorithm, one can verify correctness with the help of Lemma 5.

Theorem 6. *If the output of $\textsf{IsForbidden}(P, \{a\}, \emptyset)$ is true, then the atom a is forbidden in P.*

We are able to show for Example 5 that $\mathit{fct}(a, s(s(0)))$ is forbidden since the atom $\mathit{diff}(0, s(s(0)))$ cannot possibly have support. This would require $\neg \mathit{eq}(a, a)$, which contradicts $\mathit{eq}(a, a)$. For Example 6, it is key to notice that $r(f(b), f(f(b)))$ is forbidden.

Example 7. *We show how Algorithm 1 verifies that $r(f(b), f(f(b)))$ is forbidden in Example 6.*

- *Initialize L^+ with $r(f(b), f(f(b)))$ and L^- with $\emptyset$.*
- *In line 1, $P_\infty^-(L^+, L^-) = \{r(b, f(b))\}$; $P_\infty^+(L^+, L^-) = \{r(f(b), f(f(b))), r(a, b), \mathit{stop}(b), \mathit{stop}(f(f(b)))\}$.*
- *In the loop in line 6, pick $r(f(b), f(f(b)))$.*
- *In the loop in line 8, there is only one choice with r as the last rule, g the identity, and σ mapping Y to $f(b)$.*
- *Since g is the identity, the condition in line 9 is false.*

- *K^+ and K^- are $P^+_\infty(L^+, L^-)$ and $P^-_\infty(L^+, L^-)$.*
- *For the r-extension of σ in line 14, we pick σ' with $\sigma'(X) = c_x$ where c_x is a fresh constant. (The first position of r may feature function symbols.)*
- *In line 15, J^+ is set to K^+ extended with $r(c_x, f(b))$ and J^- is set to K^- extended with $stop(c_x)$.*
- *In the recursive call, we initialize L^+ and L^- with J^+ and J^- from outside the call.*
- *In line 1, the atom closures are $P^-_\infty(L^+, L^-) = L^-$ and $P^+_\infty(L^+, L^-) = L^+ \cup \{stop(f(b))\}$.*
- *In the loop in line 6, pick $r(c_x, f(b))$.*
- *In the loop in line 8, there is only one choice with r as the last rule, g mapping c_x to b, and σ mapping Y to b.*
- *In line 13, obtain K^+ and K^- from $P^+_\infty(L^+, L^-)$ and $P^-_\infty(L^+, L^-)$ by replacing $r(c_x, f(b))$ by $r(b, f(b))$ and $stop(c_x)$ by $stop(b)$, respectively.*
- *For the r-extension of σ in line 14, we pick σ' with $\sigma'(X) = c_x$ where c_x is again a fresh constant.*
- *In line 15, J^+ is set to K^+ extended with $r(c_x, b)$ and J^- is set to K^- extended with $stop(c_x)$.*
- *In the recursive call, we initialize L^+ and L^- with J^+ and J^- from outside the call.*
- *In line 1, the atom closures are $P^-_\infty(L^+, L^-) = L^-$ and $P^+_\infty(L^+, L^-) = L^+ \cup \{stop(b)\}$.*
- *We return true in line 3 since $stop(b) \in L^+ \cap L^-$.*

Based on these insights, we conclude that the IsConsistent check ignoring forbidden atoms according to the IsForbidden check captures Examples 5 and 6 as unsatisfiable.

5.3 Towards Finite Valid Groundings

Avoiding forbidden atoms does not only improve the IsConsistent procedure but can reduce grounding efforts.

Definition 6. *Define GroundNotForbidden$(\cdot)$ that takes a program P as input and executes the following instructions:*

1. *Initialize $i := 1$, $A_0 := \emptyset$, and $P_g := \emptyset$.*
2. *Initialize $A_i := A_{i-1}$ and for each $r \in$ Ground(P) with $B_r^+ \subseteq A_{i-1}$, do the following. If all atoms in H_r are forbidden in P, add $\leftarrow B_r$ to P_g. Otherwise, add r to P_g and add the single atom in H_r to A_i.*
3. *Stop if $A_i = A_{i-1}$; else set $i := i + 1$ and go to Step 2.*

The output of the procedure is P_g.

Observe that for any answer set I of a program P, P_g still proves all atoms in I and the rules $P_g \setminus$ Ground(P) must be satisfied by I. Vice versa, all atoms in any answer set I' for P_g are also proven in Ground(P) and I' satisfies all rules in Ground(P) as otherwise a contradiction would follow from one of the constraints in $P_g \setminus$ Ground(P).

Theorem 7. *For a program P, GroundNotForbidden(P) is a valid grounding, i.e. I is an answer set of P iff I is an answer set of GroundNotForbidden(P).*

Interestingly, the procedure always yields a finite ground program P_g for frugal and non-proliferous programs.

Proposition 5. *For a frugal and non-proliferous program P, GroundNotForbidden(P) is finite.*

Proof. By Definition 4, P has only finitely many answer sets and all of them are finite. Hence, there is only a finite number of atoms in all answer sets of P and all other atoms are forbidden. Therefore, GroundNotForbidden(P) only takes a finite number of steps to compute. $\square$

This result might be surprising but it is less so once we realize that this only holds because the definition of GroundNotForbidden assumes that we can decide if an atom is forbidden. So GroundNotForbidden is actually not computable. In practice however, any sufficient check for forbiddenness (like IsForbidden) can be used in the procedure to make it computable without sacrificing the validity of P_g, as then P_g will only contain more rules from Ground(P) and Theorem 7 still holds. This allows us to compute finite valid groundings for all of our Examples 1, 2, 5, and 6.

6 Conclusion

In this work, we have been undergoing an in-depth reconsideration of undecidability for ASP consistency. We have shown intuitive reductions from and to similarly hard problems, which can be of interest even outside of the ASP community. We also considered two characteristics of ASP programs that can make reasoning hard, that is having infinite answer sets or infinitely many answer sets. We identified *frugal* and *non-proliferous* programs as a desirable class of programs that at least ensures semi-decidability of reasoning. To cover more negative cases with a semi-decision procedure, we ignore *forbidden* atoms and show a proof-of-concept algorithm implementing a sufficient condition that captures our main examples. Furthermore, we can leverage forbidden atoms also to compute a valid grounding. In principle, GroundNotForbidden yields finite valid groundings for all frugal and non-proliferous programs if we are able to ignore all forbidden atoms. Note that, while we considered non-disjunctive programs for simplicity, our results can be extended towards disjunctions as well. This requires careful reconsiderations for Algorithm 1 but is almost immediate for all other results.

Future Research Directions and Outlook. We hope this work will reopen the discussion about function symbols and how we can design smart techniques to avoid non-terminating grounding procedures. We expect that the ASP community would benefit from incorporating recent research on termination conditions and updating their grounding strategies accordingly. After all, the ASP language is first-order based and we are convinced that function symbols are a key ingredient for elegant and convenient modeling of real-world scenarios.

An obvious future work is an efficient implementation of sufficient checks for forbidden atoms, requiring a good tradeoff between generality and performance. Our proposed procedure can function as a reference for implementation. Grounding procedures ignoring forbidden atoms can then be evaluated for existing ASP solvers. Even in lazy-grounding, termination of solvers can be improved by ignoring forbidden atoms. We think this idea is promising for improving existing reasoners like Alpha, gringo/clingo, and (i)DLV.

Acknowledgments

We want to acknowledge that the full modelling of the wolf, goat cabbage puzzle from the introduction is inspired by lecture slides created by Jean-François Baget.

On TU Dresden side, this work was partly supported by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) in project 389792660 (TRR 248, Center for Perspicuous Systems); by the Bundesministerium für Bildung und Forschung (BMBF) in the Center for Scalable Data Analytics and Artificial Intelligence (ScaDS.AI); by BMBF and DAAD (German Academic Exchange Service) in project 57616814 (SECAI, School of Embedded and Composite AI); and by the Center for Advancing Electronics Dresden (cfaed).

Carral was financially supported by the ANR project CQFD (ANR-18-CE23-0003).

Hecher is funded by the Austrian Science Fund (FWF), grants J 4656 and P 32830, the Society for Research Funding in Lower Austria (GFF, Gesellschaft für Forschungsförderung NÖ) grant ExzF-0004, as well as the Vienna Science and Technology Fund (WWTF) grant ICT19-065. Parts of the research were carried out while Hecher was visiting the Simons institute for the theory of computing at UC Berkeley.

References

[Alviano *et al.*, 2010] Mario Alviano, Wolfgang Faber, Nicola Leone, Simona Perri, Gerald Pfeifer, and Giorgio Terracina. The Disjunctive Datalog System DLV. In Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Jon Sellers, editors, *Datalog Reloaded - First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers*, volume 6702 of *Lecture Notes in Computer Science*, pages 282–301. Springer, 2010.

[Alviano *et al.*, 2012] Mario Alviano, Francesco Calimeri, Wolfgang Faber, Giovambattista Ianni, and Nicola Leone. Function Symbols in ASP: Overview and Perspectives. 2012.

[Alviano *et al.*, 2022] Mario Alviano, Carmine Dodaro, Salvatore Fiorentino, Alessandro Previti, and Francesco Ricca. Enumeration of Minimal Models and MUSes in WASP. In *LPNMR*, volume 13416 of *Lecture Notes in Computer Science*, pages 29–42. Springer, 2022.

[Banbara *et al.*, 2017] Mutsunori Banbara, Benjamin Kaufmann, Max Ostrowski, and Torsten Schaub. Clingcon: The next generation. *Theory Pract. Log. Program.*, 17(4):408–461, 2017.

[Bichler *et al.*, 2020] Manuel Bichler, Michael Morak, and Stefan Woltran. lpopt: A Rule Optimization Tool for Answer Set Programming. *Fundamenta Informaticae*, 177(3-4):275–296, 2020.

[Biere *et al.*, 2009] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[Brewka *et al.*, 2011] Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczynski. Answer set programming at a glance. *Commun. ACM*, 54(12):92–103, 2011.

[Cabalar *et al.*, 2020] Pedro Cabalar, Jorge Fandinno, Torsten Schaub, and Philipp Wanko. A uniform treatment of aggregates and constraints in hybrid ASP. In *KR*, pages 193–202, 2020.

[Calimeri *et al.*, 2008] Francesco Calimeri, Susanna Cozza, Giovambattista Ianni, and Nicola Leone. Computable functions in ASP: theory and implementation. In Maria Garcia de la Banda and Enrico Pontelli, editors, *Logic Programming, 24th International Conference, ICLP 2008, Udine, Italy, December 9-13 2008, Proceedings*, volume 5366 of *Lecture Notes in Computer Science*, pages 407–424. Springer, 2008.

[Calimeri *et al.*, 2017] Francesco Calimeri, Davide Fuscà, Simona Perri, and Jessica Zangari. I-DLV: the new intelligent grounder of DLV. *Intelligenza Artificiale*, 11(1):5–20, 2017.

[Dantsin *et al.*, 2001] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3):374–425, September 2001.

[Eiter *et al.*, 1994] Thomas Eiter, Georg Gottlob, and Heikki Mannila. Adding disjunction to datalog (extended abstract). In *PODS '94*, pages 267–278, New York, NY, USA, 1994. Assoc. Comput. Mach., New York.

[Eiter *et al.*, 2007] Thomas Eiter, Wolfgang Faber, Michael Fink, and Stefan Woltran. Complexity results for answer set programming with bounded predicate arities and implications. 51(2-4):123–165, 2007.

[Gaggl *et al.*, 2022] Sarah Alice Gaggl, Philipp Hanisch, and Markus Krötzsch. Simulating sets in answer set programming. In Luc De Raedt, editor, *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*, pages 2634–2640. ijcai.org, 2022.

[Gebser *et al.*, 2009] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Solution enumeration for projected boolean search problems. In *CPAIOR'09*, volume 5547, pages 71–86, 2009.

[Gebser *et al.*, 2012] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer Set Solving in Practice*. Morgan & Claypool, 2012.

[Gebser *et al.*, 2019] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Multi-shot ASP solving with clingo. *Theory Pract. Log. Program.*, 19(1):27–82, 2019.

[Gerlach *et al.*, 2024] Lukas Gerlach, David Carral, and Markus Hecher. Finite groundings for asp with functions: A journey through consistency (technical report). https://iccl.inf.tu-dresden.de/web/Inproceedings3380, 2024.

[Harel, 1986] David Harel. Effective transformations on infinite trees, with applications to high undecidability, dominoes, and fairness. *J. ACM*, 33(1):224–248, 1986.

[Hippen and Lierler, 2021] Nicholas Hippen and Yuliya Lierler. Estimating grounding sizes of logic programs under answer set semantics. In *JELIA*, volume 12678, pages 346–361. Springer, 2021.

[Janhunen and Niemelä, 2016] Tomi Janhunen and Ilkka Niemelä. The answer set programming paradigm. *AI Mag.*, 37(3):13–24, 2016.

[Janhunen *et al.*, 2017] Tomi Janhunen, Roland Kaminski, Max Ostrowski, Sebastian Schellhorn, Philipp Wanko, and Torsten Schaub. Clingo goes linear constraints over reals and integers. *Theory Pract. Log. Program.*, 17(5-6):872–888, 2017.

[Kaminski and Schaub, 2021] Roland Kaminski and Torsten Schaub. On the foundations of grounding in answer set programming. *CoRR*, abs/2108.04769, 2021.

[Kleine Büning and Lettman, 1999] Hans Kleine Büning and Theodor Lettman. *Propositional logic: deduction and algorithms*. Cambridge University Press, Cambridge, 1999.

[Marek and Remmel, 2011] Victor Marek and Jeffrey B. Remmel. Effectively Reasoning about Infinite Sets in Answer Set Programming. In Marcello Balduccini and Tran Cao Son, editors, *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning: Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday*, Lecture Notes in Computer Science, pages 131–147. Springer, Berlin, Heidelberg, 2011.

[Marek and Truszczyński, 1991] Wiktor Marek and Mirosław Truszczyński. Autoepistemic logic. *J. of the ACM*, 38(3):588–619, 1991.

[Marek *et al.*, 1994] V. Wiktor Marek, Anil Nerode, and Jeffrey B. Remmel. The Stable Models of a Predicate Logic Program. *The Journal of Logic Programming*, 21(3):129–154, November 1994.

[Rogers, 1987] Hartley Rogers, Jr. *Theory of recursive functions and effective computability (Reprint from 1967)*. MIT Press, 1987.

[Weinzierl *et al.*, 2020] Antonius Weinzierl, Richard Taupe, and Gerhard Friedrich. Advancing Lazy-Grounding ASP Solving Techniques - Restarts, Phase Saving, Heuristics, and More. *Theory Pract. Log. Program.*, 20(5):609–624, 2020.